Knight’s Tour

Julia Tiemi

Trabalho Prático 1 de Programação Modular

Universidade Federal de Minas Gerais

2018/2

1. Parte I - Relatório sobre o Projeto

A primeira parte deste documento consiste num relatório sobre o problema e a implementação feita para resolvê-lo.

1.1 **Introdução**

Knight's Tour (em português, Passeio do Cavalo) é um problema matemático que é classicamente resolvido por meio de um programa computacional.

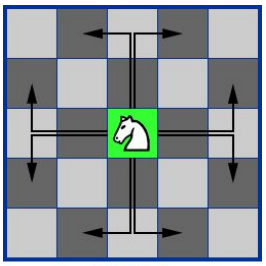

Figura 1

No xadrez, temos um tabuleiro 8x8 e uma das peças do jogo é o cavalo. O movimento dessa peça envolve, a partir de uma casa, forma um L. Ou seja, andar duas casas em uma direção e mais uma na direção perpendicular à anterior. É melhor visualizado na Figura 1.

O problema do Knight's Tour consiste em fazer com que o cavalo percorra todas as posições do tabuleiro a partir de uma determinada casa, passando por cada casa apenas uma vez.

Na implementação realizada para este projeto, o tabuleiro utilizado foi um clássico 8x8 tal qual o real tabuleiro de xadrez. A posição inicial foi determinada randomicamente. O modelo de passeio escolhida para o projeto foi a de Backtracking.

Backtracking é um paradigma de programação onde todas as possíveis soluções são testadas até achar uma que resolva o problema em questão. Ela será melhor discutida na metodologia. No Knight's Tour, ela funciona da seguinte maneira: começando com o tabuleiro vazio, passo a passo construímos o passeio do cavalo. Se o cavalo não pode mais se mover e o passeio não foi completado, voltamos (backtrack) até o último ponto válido e tentamos um novo movimento. Se voltarmos até o ponto inicial após tentadas todas as possibilidades, chegamos a conclusão de que não há solução.

1.2 **Implementação**

O projeto foi dividido em cinco classes, todas sob o mesmo package KnightsTour. Isso facilitou a reutilização dos métodos e classes. Serão detalhados a seguir os métodos e responsabilidades de cada uma, explicando sobre a lógica dessa divisão.

1.2.1 Enumerations

Esta classe contém um ***enum movementDirection*** que determina cada possível movimento do cavalo. Cada valor tem um dupla de valores, determinando seu deslocamento na direção y (entre as linhas da matriz) e x (entre as colunas). No construtor desse enum, atribuímos essa dupla a ***nextRowDistance*** e ***nextColumnDistance*** para podermos referenciar a cada direção separadamente durante o programa.

1.2.1 Position

Esta classe contém apenas duas propriedades: ***row*** e ***column***. Isso foi apenas uma forma de utilizar coordenadas organizadamente.

1.2.3 Board

Essa classe representa o tabuleiro e contém três propriedades. Duas constantes que representam o tamanho do tabuleiro - ***DIMENSION*** que é o tamanho do lado do tabuleiro e ***SIZE*** que é o total de casas - e uma matriz de duas dimensões

chessBoard. O construtor dessa classe constrói a matriz DIMENSION x DIMENSION preenchida com 0's.

Seu primeiro método é ***public Position decideInitialKnightPosition()*** que determina onde o cavalo séra posicionado. Com ajuda da biblioteca *java.util.Random*, ele randomiza as posições iniciais para que sempre estejam entre 0 e 7 (DIMENSION possibilidades) e as retorna no formato de uma ***Position***.

O método ***public void registerMovement(Position position, int movementNumber)*** registra um movimento feito pelo cavalo para uma casa ***position*** e com a contagem de movimentos ***movementNumber***.

O método ***public void unregisterMovement(Position position)*** apaga um movimento feito que não mostrou resposta válida após encurralar o cavalo na casa ***position*** sem terminar o passeio.

O último método desta classe é ***public void printFinishedBoard()*** que imprime o tabuleiro com a movimentação do cavalo após a resposta ser encontrada.

1.2.4 Knight

Esta classe representa a peça de xadrez cavalo. Ela contém duas propriedades, uma ***Position position*** que representa em qual casa o cavalo está num momento exato e um contador ***numberOfMovements*** que conta quantos passos o cavalo deu até um exato momento.

No construtor da classe, definimos o número inicial de movimentos, zero.

O primeiro método da classe é ***public void placeKnight(Position position)***, que recebe uma posição (calculada em ***decideInitialKnightPosition()***) e posiciona o cavalo na casa de onde ele iniciará seu passeio.

O método ***public void startTour(Position position, Board board)*** recebe a posição inicial do cavalo e o tabuleiro vazio e começa a rodar o algoritmo recursivo. Se ele falhar, ele informa que o passeio não foi possível.

O método ***public boolean isAlreadyStepped(Board board, Position position)*** checa, dada uma posição e um estado do tabuleiro, se o cavalo já esteve por ali. Isso não pode ocorrer porque faz parte da solução que o cavalo tenha passado por cada casa apenas uma vez.

O método ***public boolean isMoveAvailable(Position position, int boardDimension)*** checa se o movimento a se fazer está dentro dos limites do tabuleiro.

O método ***public void countSteps()*** adiciona em um a contagem de passos do cavalo.

O método ***public void undoStep(Position position, Board board)*** desfaz o movimento caso ele se mostre incapaz de chegar na solução. Ou seja, quando o cavalo chega em uma posição que ele não havia estado ainda, mas não consegue se movimentar depois disso para uma outra posição nova, e seu passeio ainda não chegou ao fim, ele precisa retroceder (segundo o algoritmo de Backtracking). Isso será melhor discutido na próxima seção.

O método ***public boolean isFinishedMoving(int size)*** checa se o cavalo terminou seu passeio, ou seja, no caso de um tabuleiro 8x8, se ele deu 64 passos (um em cada casa).

O método ***public boolean move(Position position, Board board)*** é o algoritmo que resolve o problema de fato. Ele parte de uma abordagem recursiva utilizando Backtracking. Primeiramente, ele checa se a casa que passada pela ***position*** já foi visitada. Se sim, ele não pode estar ali e termina, voltando para a execução anterior da função.

Depois, ele registra essa nova posição do cavaleiro e escreve qual o número desse passo no tabuleiro. Se esse for o movimento final, ele termina e apresenta a solução encontrada. Se não, ele continua a recursividade chamando o próximo método que será explicado a seguir para cada uma das possíveis direções que ele pode executar seu movimento. Se nenhuma dessas opções for uma caminhada válida, esta casa que ele está não consegue continuar a movimentação, então ele deve retroceder até o ponto de poder continuar o passeio.

public boolean checkNextMove(Board board, Enumerations.movementDirection direction, Position position) é o método que checa se a intenção do movimento pode ou não ser realizada. Checando, primeiramente se o movimento será executado dentro dos limites do tabuleiro, e depois se nessa próxima posição ele continuará o movimento (recursão com ***move()***).

1.2.5 Knights Tour

Contém a main do programa. Ela tem como função apenas instanciar as classes Knight e Board - que ainda serão descritas - e rodar o algoritmo de resolução. Não contém nenhum método ou propriedade próprio.

1.3 **Resultados e Discussão**

A seguir serão apresentados uma série de resultados de diferentes posições iniciais sorteadas:

```
Initial Position: [0][3]
Total number of Movements: 18524326
  12  03  10  01  56  39  54  35
  09  26  13  38  15  36  17  40
  04  11  02  57  32  55  34  53
  27  08  25  14  37  16  41  18
  24  05  58  31  62  33  52  49
  59  28  07  46  21  50  19  42
  06  23  30  61  44  63  48  51
  29  60  45  22  47  20  43  64
CONSTRUÍDO COM SUCESSO (tempo total: 1 segundo)
```

```
Initial Position: [3][5]
Total number of Movements: 686933791
  17  56  15  20  07  24  03  22
  36  45  18  57  02  21  08  25
  55  16  35  14  19  06  23  04
  46  37  44  39  58  01  26  09
  43  54  59  34  13  32  05  30
  60  47  38  51  40  29  10  27
  53  42  49  62  33  12  31  64
  48  61  52  41  50  63  28  11
CONSTRUÍDO COM SUCESSO (tempo total: 39 segundos)
```

Initial Position: [3][5]
Total number of Movements: 686933791
17 56 15 20 07 24 03 22
36 45 18 57 02 21 08 25
55 16 35 14 19 06 23 04
46 37 44 39 58 01 26 09
43 54 59 34 13 32 05 30
60 47 38 51 40 29 10 27
53 42 49 62 33 12 31 64
48 61 52 41 50 63 28 11
CONSTRUÍDO COM SUCESSO (tempo total: 41 segundos)

Initial Position: [5][3]
Total number of Movements: 442906998
21 10 25 12 27 14 43 16
24 03 22 19 42 17 38 29
09 20 11 26 13 28 15 44
04 23 02 41 18 37 30 39
59 08 57 64 53 40 45 36
56 05 60 01 48 33 52 31
61 58 07 54 63 50 35 46
06 55 62 49 34 47 32 51
CONSTRUÍDO COM SUCESSO (tempo total: 25 segundos)

Initial Position: [2][7]
Total number of Movements: 54672721
08 13 06 49 04 59 02 51
11 30 09 58 19 50 21 60
14 07 12 05 48 03 52 01
31 10 29 18 57 20 61 22
28 15 32 47 62 43 56 53
35 38 17 40 25 54 23 44
16 27 36 33 46 63 42 55
37 34 39 26 41 24 45 64
CONSTRUÍDO COM SUCESSO (tempo total: 3 segundos)

```
Initial Position: [6][0]
Total number of Movements: 51855978
  29  04  31  06  43  08  55  10
  32  15  28  13  26  11  44  57
  03  30  05  42  07  56  09  54
  16  33  14  27  12  25  58  45
  37  02  39  20  41  22  53  24
  34  17  36  63  50  61  46  59
  01  38  19  40  21  48  23  52
  18  35  64  49  62  51  60  47
CONSTRUÍDO COM SUCESSO (tempo total: 3 segundos)
```

É fácil perceber como diferentes posições iniciais mudam o desempenho do programa. É muito difícil prever o comportamento do programa, pois até a ordem da checagem dos movimentos altera a quantidade total de passos e o tempo de execução.

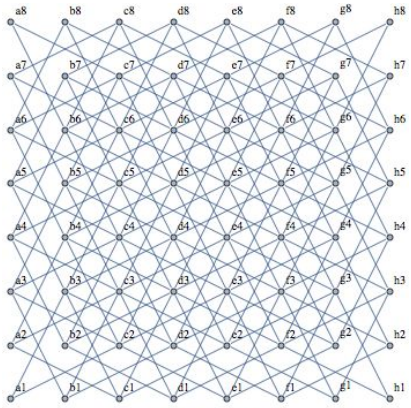

Figura 2

O Backtracking não é a melhor solução para esse problema. Num tabuleiro 8x8, existem um total de 26,534,728,821,064 possíveis passeios. O cavalo deve encontrar o seu passeio antes disso, mas muitas opções podem ser testadas antes do cavalo obter seu caminho. Uma abordagem melhor seria pelo algoritmo do Caminho Hamiltoniano.

Ele propõe passar por todos os vértices de um grafo exatamente uma vez. No passeio do cavalo, ele faz com que as possibilidades de caminho sejam brutalmente reduzidas, utilizando um grafo como o da Figura 2.

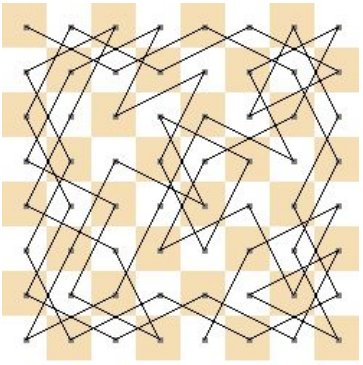

Figura 3

Outra possibilidade seria o uso de uma heurística. Existe um algoritmo proposto por Warnsdorff em 1823 que acha o caminho do passeio sem nenhum backtrack. Ele computa qual deve ser seu próximo movimento dando nota para cada possibilidade, representado na Figura 3.

1.4 **Conclusão**

Apesar de prover a solução correta, o algoritmo de backtracking não é a melhor escolha para o problema do Passeio do Cavalo. As diferentes configurações alteram muito o tempo necessário para finalizar o programa. O tempo de execução do algoritmo é muito sensível ao estado inicial de onde o cavalo é posicionado.

2. Parte II - Pesquisa

Na fase de design de um novo software, decide-se uma solução para o SRS - Software Requirements Specification, em português, Especificação de Exigência de Software - que é uma descrição do software a ser desenvolvido. Ao final dessa fase, temos o SDD - Software Design Documento, em português, Documento de Design de Software.

A fase de design envolve informar o cliente o que o conceito de software pensado se propõe a fazer e compreender como desenvolvedor quais são o hardware e software necessários para atender a demanda do cliente a partir do design técnico.

Ao fazer o design de um software, o programa é dividido em módulos. A modularização permite que cada pequena parte de um programa possa ser independente, facilitando a compreensão do programa, tornando sua manutenção mais fácil e permitindo a reutilização de partes do código.

Para medir o nível de interdependência entre os módulos, existe um conceito chamado Acoplamento. Quanto menor o acoplamento, melhor o software. Existem vários tipos de acoplamento:

Acoplamento de Conteúdo define se o conteúdo de um módulo pode ser alterado dentro de outro módulo ou se o fluxo de controle passa de um módulo para outro.

Acoplamento Comum checa se os módulos possuem dados compartilhados como estruturas de dados globais. Isso é um problema porque, ao alterar um desses dados, .é muito difícil avaliar o efeito da mudança em cada módulo em que eles estão. Isso reduz a reusabilidade dos módulos, a facilidade de manutenção e o controle no acesso dos dados.

Acoplamento de Controle determina se os módulos se comunicam trocando informação de controle. Se parâmetros passados em uma função, por exemplo, permitem um comportamento completamente diferente, isso é muito ruim. Porém, se isso apenas torna a função mais reutilizável e mais fatorada, então pode não ser um problema.

Acoplamento de Carimbo, toda a estrutura de dados e passada de um módulo para o outro. Envolve o conceito de *tramp data*, que são dados que são passados de uma função para a outra, sem ter sido usada na primeira (além de executar essa passagem). Afeta desempenho.

Acoplamento de Dados existe se os módulos são independentes entre si e se comunicam através de dados.

Outro conceito para classificar o quão bom é um software é o de Coesão. Ele mede o nível em que os elementos de um módulo estão relacionados funcionalmente. Mede em em qual quantidade os elementos que tem como função cumprir uma determinada única tarefa estão contidos em um mesmo componente. Quanto maior a coesão, melhor o software.

Coesão Lógica diz sobre os elementos estarem ligados pela lógica, e não pela funcionalidade.

Coesão Temporal diz sobre os elementos estarem ligados pelo fator temporal, ou seja, se as tarefas de um módulo são executadas no mesmo intervalo de tempo. Esse tipo de coesão contém o código que inicializa as partes do sistema.

Coesão Processual asseguram a ordem de execução. As tarefas estão fracamente ligadas e provavelmente não serão reutilizadas.

Coesão Sequencial determina se um método gera dados que serão entrada em outro método. Isso checa se o fluxo de dados está fracionado.

Coesão Funcional checa se todos os elementos para uma tarefa estão dentro do módulo.

3. Anexo - Código

```
package knightstour;

/**
 *
 * @author juliatiemi
 */
public class Enumerations {
    public enum movementDirection {
        /**
         * . 1 . . .
         * . . . . .
         * . . 0 . .
         * . . . . .
         * . . . . .
         */
        topLeft (-2, -1),
        /**
         * . . . 1 .
         * . . . . .
         * . . 0 . .
         * . . . . .
         * . . . . .
         */
        topRight (-2, 1),
        /**
         * . . . . .
         * . . . . .
         * . . 0 . .
         * . . . . .
         * . 1 . . .
         */
        bottomLeft (2, -1),
        /**
         * . . . . .
         * . . . . .
         * . . 0 . .
         * . . . . .
         * . . . 1 .
         */
        bottomRight (2, 1),
        /**
         * . . . . .
         * 1 . . . .
         * . . 0 . .
         * . . . . .
         * . . . . .
         */
```

```
        midTopLeft (-1, -2),
        /**
         * . . . . .
         * . . . . 1
         * . . 0 . .
         * . . . . .
         * . . . . .
         */
        midTopRight (-1, 2),
        /**
         * . . . . .
         * . . . . .
         * . . 0 . .
         * 1 . . . .
         * . . . . .
         */
        midBottomLeft (1, -2),
        /**
         * . . . . .
         * . . . . .
         * . . 0 . .
         * . . . . 1
         * . . . . .
         */
        midBottomRight (1, 2);

        /**
         *
         * Constructor
         */
        movementDirection(int rows, int columns) {
            nextRowDistance = rows;
            nextColumnDistance = columns;
        }

        /**
         * Represents how many squares the knight has to move vertically and
horizontally.
         */
        public final int nextRowDistance;
        public final int nextColumnDistance;

    }
}
```

```
package knightstour;

/**
 *
 * @author juliatiemi
 */

/**
 *
 * Represents coordinates
 */
public class Position {
    int row;
    int column;
}
```

```
package knightstour;

import java.text.DecimalFormat;
import java.util.Random;

/**
 *
 * @author julia
 */


/**
 *
 * DIMENSION determinates
 */
public class Board {
    public static final int DIMENSION = 8;
    public static final int SIZE = DIMENSION * DIMENSION;
    public int[][] chessBoard;

    public Board() {
        this.chessBoard = new int[DIMENSION][DIMENSION];
        for(int i = 0; i < DIMENSION; i++) {
            for(int j = 0; j < DIMENSION; j++) {
                this.chessBoard[i][j] = 0;
            }
        }
    }

    public Position decideInitialKnightPosition() {
        Position initialPosition = new Position();
        Random rng = new Random();
        int index = rng.nextInt();
        initialPosition.row = index < 0 ? (index*-1)%DIMENSION :
index%DIMENSION;
        index = rng.nextInt();
        initialPosition.column = index < 0 ? (index*-1)%DIMENSION :
index%DIMENSION;
        System.out.print("Initial Position: [" + initialPosition.row + "][" +
initialPosition.column + "]");
        System.out.println();
        return initialPosition;
    }

    public void registerMovement(Position position, int movementNumber) {
        this.chessBoard[position.row][position.column] = movementNumber;
    }

    public void unregisterMovement(Position position) {
        this.chessBoard[position.row][position.column] = 0;
    }
```

```
    public void printFinishedBoard() {
        DecimalFormat twoDigitsNumber = new DecimalFormat("00");
        for(int i = 0; i < this.DIMENSION; i++) {
            for(int j = 0; j < this.DIMENSION; j++) {
                System.out.print("   " +
twoDigitsNumber.format(this.chessBoard[i][j]));
            }
            System.out.println();
        }
    }
}
```

```
package knightstour;

/**
 *
 * @author juliatiemi
 */
public class Knight {


    /**
     *
     * Position shows where the knight is within the board
     * numberOfMovements shows how many steps the knight has taken until the
current position
     */
    public Position position;
    public int numberOfMovements;
    public int numberOfAllMovements;

    /**
     *
     * When a knight is created, it has not started walking
     */
    public Knight() {
        this.position = new Position();
        this.numberOfMovements = 0;
        this.numberOfAllMovements = 0;
    }

    /**
     *
     * @param position is the initial position randomized before and where
the knight is going to begin his tour
     */
    public void placeKnight(Position position) {
        this.position.row = position.row;
        this.position.column = position.column;
    }

    /**
     *
     * @param position is the initial position randomized before
     * @param board the clean board
     */
    public void startTour(Position position, Board board) {
        if(move(position, board)) {
            board.printFinishedBoard();
        }
        else {
            System.out.println("I cannot finish mine own toureth from this
starteth pointeth!");
```

```
        }
    }

    /**
     *
     * @param board is the actual state of the board
     * @param position is the position the knight has just stepped in
     * @return true or false, indicating if the knight has already stepped in
this square
     */
    public boolean isAlreadyStepped(Board board, Position position) {
        if(board.chessBoard[position.row][position.column] != 0) {
            return true;
        }
        else {
            return false;
        }
    }

    /**
     *
     * @param position is the position the knight has just stepped in
     * @param boardDimension is the length of the board
     * @return true or false, indicating if the move is within the board
     */
    public boolean isMoveAvailable(Position position, int boardDimension) {
        if(position.row >= 0 && position.column >= 0 &&
           position.row < boardDimension && position.column < boardDimension)
{
            return true;
        }
        else {
            return false;
        }
    }

    /**
     * Adds in one the number of steps taken by the knight
     */
    public void countSteps() {
        this.numberOfMovements = this.numberOfMovements + 1;
        this.numberOfAllMovements = this.numberOfAllMovements + 1;
    }

    /**
     *
     * @param position is the position the knight has just stepped in
     * @param board is the actual state of the board, given that the knight
made a step that did not work
     *
```

```
     * Subtract in one the number of steps taken by the knight, given that
the knight made a step that did not work
     */
        public void undoStep(Position position, Board board) {
        this.numberOfMovements = this.numberOfMovements - 1;
        board.unregisterMovement(position);
    }

    /**
     *
     * @param size the number of squares in the board
     * @return is the knight has made the same number of moves as the number
of squares in the board a.k.a has finished the tour
     */
    public boolean isFinishedMoving(int size) {
        if(this.numberOfMovements == size) {
            System.out.println("Total number of Movements: " +
this.numberOfAllMovements);
            System.out.println();
            return true;
        }
        else {
            return false;
        }
    }

    /**
     *
     * @param position is the position the knight has just stepped in
     * @param board is the actual state of the board
     * @return true or false, indicating if the move was made succesufully or
if the knight will have to backtrack
     */
    public boolean move(Position position, Board board) {

        //Have I already stepped in this square?
        if(isAlreadyStepped(board, position)) {
            return false;
        }

        //No? Great. Which step is this?
        this.countSteps();
        board.registerMovement(position, this.numberOfMovements);

        //Is this my final step for completing my tour?
        if(this.isFinishedMoving(board.SIZE)) {
            return true;
        }

        //No? So lets continue my walk
```

```
        if(checkNextMove(board, Enumerations.movementDirection.topLeft,
position)) {
            return true;
        }
        if(checkNextMove(board, Enumerations.movementDirection.topRight,
position)) {
            return true;
        }
        if(checkNextMove(board, Enumerations.movementDirection.midTopRight,
position)) {
            return true;
        }
        if(checkNextMove(board,
Enumerations.movementDirection.midBottomRight, position)) {
            return true;
        }
        if(checkNextMove(board, Enumerations.movementDirection.bottomRight,
position)) {
            return true;
        }
        if(checkNextMove(board, Enumerations.movementDirection.bottomLeft,
position)) {
            return true;
        }
        if(checkNextMove(board, Enumerations.movementDirection.midBottomLeft,
position)) {
            return true;
        }
        if(checkNextMove(board, Enumerations.movementDirection.midTopLeft,
position)) {
            return true;
        }

        //Hmm, can´t keep going. I shall move backwards and retry.
        undoStep(position, board);
        return false;
    }

    /**
     *
     * @param board is the actual state of the board
     * @param direction is which of the eight possible knight's moviment he
is trying to do
     * @param position is the position the knight has just stepped in
     * @return true or false, indicating if the next move is a possibility
     */
    public boolean checkNextMove(Board board, Enumerations.movementDirection
direction, Position position) {
        position.row = position.row + direction.nextRowDistance;
        position.column = position.column + direction.nextColumnDistance;
```

```
        if(isMoveAvailable(position, board.DIMENSION) && move(position,
board)) {
            return true;
        }
        else {
            position.row = position.row - direction.nextRowDistance;
            position.column = position.column - direction.nextColumnDistance;
            return false;
        }
    }
}
```

```
package knightstour;

/**
 *
 * @author julia
 */
public class KnightsTour {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        Board board = new Board();
        Knight knight = new Knight();

        Position initialPosition = board.decideInitialKnightPosition();

        knight.placeKnight(initialPosition);

        knight.startTour(initialPosition, board);
    }

}
```